

# USB in Embedded Design (Part 1)

## The Undeniable Benefits

*How much do you really know about USB? This column is a great place to start educating yourself about the basics. And it couldn't have come at a better time. Serial and parallel ports may soon be things of the past.*

**T**hey tell you: "Prepare; the end is near. For those who are not prepared shall not enter the kingdom of the PC."

Where would your project be without access to today's GUI power base? Like it or not, PCs rule. For many a designer, they are *numero uno*, an important piece of the puzzle, and I'm not talking about a PC as a development platform.

OK, so maybe your device doesn't need that free user GUI connection that's already in every home. However, I'd find it hard to believe that you've never designed a gizmo that didn't use a serial or parallel port. Most of you have.

And now they're telling you, "You won't find those on new PC designs."

You've already seen the appearance of thin USB ports on the back, front, and side of towers and laptops. Soon the legacy serial and parallel ports will vanish completely from your computers. Whether you believe this prediction or not, it would be wise to prepare for the possibility. But don't let the promised extinction scare you, there are other reasons for you to embrace USB with open ports—eh, arms.

### WISH LIST

From a user's point of view, I want to be able to connect all of my peripher-

als via the same cable, which means using the same connector on all products. I want to be able to plug stuff in at any time (hot pluggable) and then have the computer recognize it and automatically load whatever driver is necessary without restarting the machine. I would also like to eliminate the tangle of power supply dongles needed to externally power every peripheral.

As a designer, I don't want to have to make choices about the physical interface, serial or parallel, DB25 or DE9, and male or female. I'd be happier not having to support different data formats. Having to deal with extra level-shifting circuitry only adds to the cost.

### USB IMPLEMENTERS FORUM

To truly come up with a standard that is embraced by everyone requires that many different parties have input in the specification process. Presently, seven corporations are assigned the copyright to the USB standard: Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. A visit to [www.usb.org](http://www.usb.org) will show you the level of commitment to the USB push.

Windows 95 (SR2) supported the 1996 release of USB 1.0, although it was not until Windows 98 (SE) that the hardware and software issues were under control. A faster USB 2.0 was introduced in 2000, thanks partly to a head-to-head competition with Firewire. The first USB on-the-go specifications (device as a host) were released a year later.

### USB

The USB allows data exchange between a host and peripherals (devices). All USB devices share the bandwidth through a host-scheduled, token-based protocol, which allows each device to be attached, configured, used, and detached while the system remains in operation. This requires three basic ingredients: a host, a device, and an interconnect.

The host, or controller, must detect the attachment and removal of devices (enumeration and configuration),

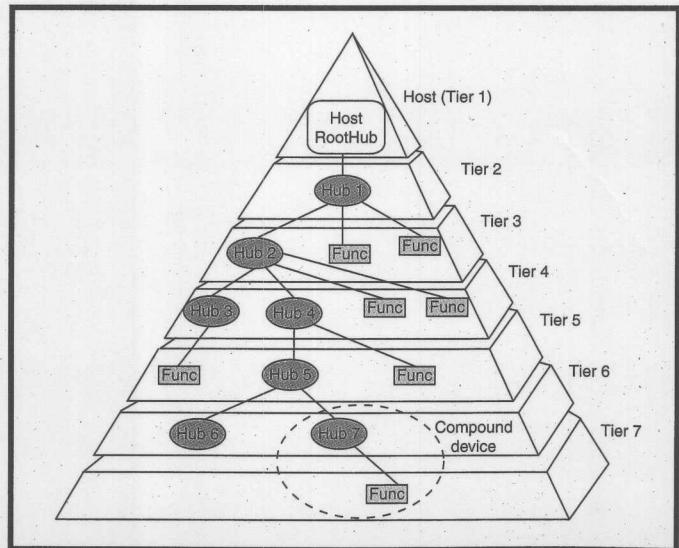


Figure 1—The USB connection topography is a one-to-one connection between the host and a device. The hub is a specialized device acting as a repeater to enable multiple one-to-one connections through a tiered system.

manage the flow of control and data packets (isochronous and asynchronous), collect status (device and bus), and provide power management to the devices. A device provides special capabilities to the system and may contain multiple pipes (communications channels) through which the host may communicate with the device.

Each device must contain a control pipe connected to endpoint zero, a common access mechanism for accessing information about the device (like a table of contents). Endpoint zero contains standard information like vendor ID, device class, and power management capabilities, as well as device configuration, interface, and endpoint descriptors. Endpoint zero also contains class and vendor-specific information. Although the USB specification makes a distinction between a device and a hub, you can think of the hub as just another specialized device.

Let's look at the way USB connections are made to understand why the hub is considered so important. Figure 1 shows a seven-tier network of USB connections made possible by the hub. Between tier 1 (a host) and tier 2, you have room for one USB device. After a connection is made between

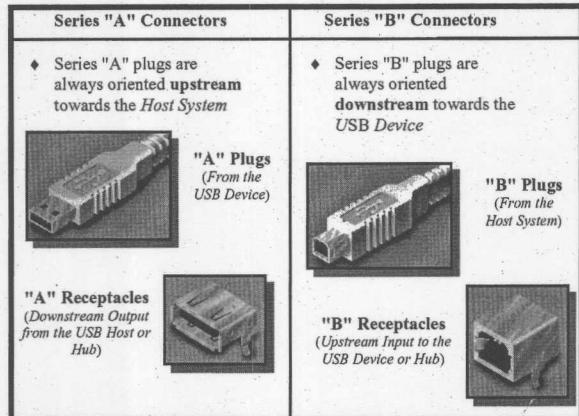


Photo 1—Every USB cable is wired the same. Never again will you need to search for a cable with the right connectors.

the host and a device, there is no room for other connections. This allows only a single device to be connected to a host. Not too useful, eh?

Like all devices, a hub has one connection upstream toward the host. Yet a hub offers many connections downstream to other devices. (Most PCs have a built-in hub, offering you more than one USB connection.) Every device connected to a hub (or the host) operates from a lower tier. The USB specs allow devices to operate down to tier 7 (hubs to tier 6).

The USB hub has three jobs: the hub controller, the hub repeater, and the transaction translator (see Figure 2). The hub controller controls the upstream (hub to host) communication path and can configure and monitor its downstream (hub to device) ports. The hub repeater's job is to control communications between the upstream port and the downstream ports, including hardware support for reset and suspend/resume signaling. The transaction translator handles communication speed differences between the upstream and downstream paths. (This allows each path to take full advantage of its highest operating speed.)

Each device (including hubs) is given its own address during enumeration. The host uses this address as a way of talking to one device at a time. A host broadcasts packets, which carry a device address to the entire bus. The packet burrows through every pyramid tier thanks to the hub's ability to retransmit the packet to every downstream device (including

translation between bus segments of different speeds). Eventually, all devices get to hear the packet. However, only the addressed device will respond. This means one return packet (at most) is generated and works its way back up through any tiers to the host.

## INTERFACE

With the introduction of USB 2.0, the bus has three possible communication speeds: low speed (1.5 Mbps), full speed (12 Mbps), and high speed (480 Mbps).

Although the specifications may differ on the transceivers and cable used for the implementation of each, this is all invisible to you. You may not even know which one each device is using. Hint: Although a device may be capable of high-speed communication, unless all of the upstream hubs support high-speed communication, the interface will throttle back to the highest supported speed. So it makes sense to know the capabilities of each device and hub and to connect them in a way that maximizes bandwidth.

The physical interface is a four-wire connection for all. Separate upstream (host) and downstream (peripheral) connectors are used. These keyed connectors ensure the cabling is correct and the signal connections are made in sequence (power first). Photo 1 shows both plugs and receptacles for upstream and downstream use. The signal connections are shown in Table 1. Data is sent in serial fashion using differential pair drivers and receivers (D+ and D-). The cable specifications require that the data pair be twisted to obtain the full benefit of using differential signals to cancel common mode interference.

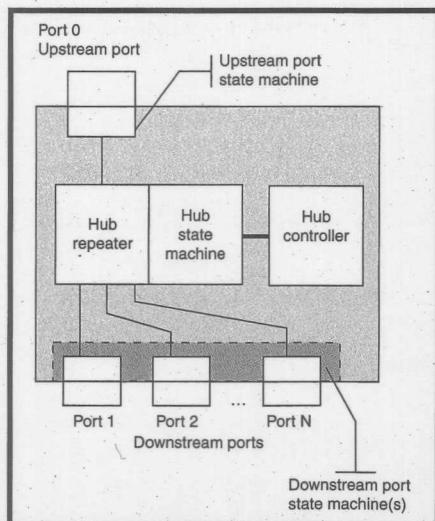


Figure 2—The hub is a special device acting as a repeater, controller, and transaction translator. It establishes communication with downstream devices at the highest possible speed. It can receive packets at one speed and send them at a different speed to maximize bandwidth.

Contact number	Signal name	Typical wiring assignment
1	VBUS	red
2	D-	White
3	D+	Green
4	GND	Black
Shell	Shield	Drain wire

Table 1—Connecting USB devices requires a cable having different upstream and downstream connectors. The USB system is designed to be hot-pluggable.

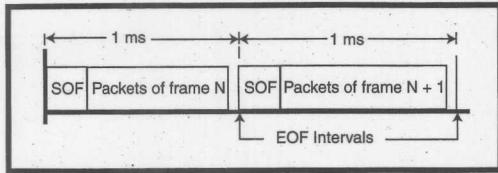


Figure 3—The host creates 1-ms transfer frames, which can carry packets to any USB device on the bus. Packets contain ID information that allows the packet to be directed down a pipe (logical connection) to a single device's endpoint. Data packets can be formatted or non-formatted depending on the type of transfer used.

But how is the maximum speed for each path determined if all of the connectors and connections are the same? A device on a lower tier is responsible for declaring its maximum speed to the host (or hub) it is connected to. In USB 1.0, a pull-up resistor tied to D- raises this signal slightly off of an idle state (with reference to D+), indicating a low-speed connection, while a pull-up tied to the D+ indicates a device capable of a full-speed connection.

High speed was added to USB 2.0. To stay compatible with the previous USB 1.0 specification, a device must first be declared full-speed and only then will an upstream high-speed transceiver begin a high-speed test sequence. The downstream device must interpret the test and respond indicating to the upstream port that it is a high-speed device. The full-speed pull-up is then disabled (from D+) on the device and special high-speed transceivers are enabled on the hub. Any further communication within this segment of the bus takes place at high speed.

## DATA

To reduce the number of signals needed for USB, a clock is encoded along with the data into an NRZI scheme using bit stuffing, which is the automatic addition of one 0 bit after six consecutive 1 bits. This is required to force a change-of-state in the NRZI encoded data allowing a receiver to remain in sync with the data. The receiver automatically discards this bit.

USB communication can be divided into two categories—configuration and data exchange. The former is covered in the next section.

Data exchange is handled using 1-ms frames (see Figure 3). (Note that high-speed frames are further divided into

8- to 125- $\mu$ s microframes.)

During each frame, the host may send transaction packets to any or all devices on the bus. Each frame may be filled with packets up to the 1-ms frame time. A host transaction may be one of four types: control, bulk, interrupt, and isochronous. Control transfers are normally used for configuration at the time of attachment, but can be used for other device-specific purposes. Bulk transfers are used for bursty traffic that isn't time critical, as in a file transfer. Interrupt transfers are used for passing reliable data in real time. Isochronous data transfers stream time scheduled packets and can be bandwidth hogs. The use of timed frames allows the host to guarantee time on

the bus for certain transactions.

Depending on the type, each transaction consists of two or three phases: token, data, and, for all but isochronous types, handshaking. All packets begin with a packet-ID (PID) identifying the type and phase, and may contain other information such as the USB device address and endpoint number. The token phase packet includes information on how to treat the following data as setup, data in, data out, or status. The data phase transfers data to or from the device based on information from the token phase. A handshaking phase may indicate the status of a transaction using an ACK, NAK, STALL, or NYET code.

A transaction connection between the host and device is called a pipe. Each pipe reflects the characteristics

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of this descriptor in bytes
1	bDescriptorType	1	Constant	Descriptor type
2	bcdUSB	2	BCD	USB specification. Release number in binary-coded decimal (i.e., 2.10 is 210H). This field identifies the release of the USB specification with which the device and its descriptors are compliant.
4	BdeviceClass	1	Class	Class code (assigned by the USB-IF). If this field is reset to zero, each interface within a configuration specifies its own class information, and the various interfaces operate independently. If this field is set to a value between one and FEH, the device supports different class specifications on different interfaces, and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to FFH, the device class is vendor-specific.
5	BdeviceSubClass	1	SubClass	Subclass code (assigned by the USB-IF). These codes are qualified by the value of the bDeviceClass field. If the bDeviceClass field is reset to zero, this field also must be reset to zero. If the bDeviceClass field is not set to FFH, all values are reserved for assignment by the USB-IF.
6	bDeviceProtocol	1	Protocol	Protocol code (assigned by the USB-IF). These codes are qualified by the value of the bDeviceClass and the bDeviceSubClass fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use class-specific protocols on a device basis. However, it may use class-specific protocols on an interface basis. If this field is set to FFH, the device uses a vendor-specific protocol on a device basis.
7	bMaxPacketSize0	1	Number	Maximum packet size for endpoint zero (only 8, 16, 32, or 64 are valid)
8	idVendor	2	ID	Vendor ID (assigned by the USB-IF)
10	idProduct	2	ID	Product ID (assigned by the manufacturer)
12	bcdDevice	2	BCD	Device release number in binary-coded decimal
14	iManufacturer	1	Index	Index of string descriptor describing manufacturer
15	iProduct	1	Index	Index of string descriptor describing product
16	iSerialNumber	1	Index	Index of string descriptor describing the device's serial number
17	bNumConfigurations	1	Number	Number of possible configurations

Table 2—This is the format of the device descriptor table. As the first piece of information that a host requests, it contains a basic description of the device and how the host can best communicate with it.

of its associated device endpoint (i.e., bulk in using endpoint one). The default endpoint zero must always exist on a powered device to provide initial device configuration, status, and control information during the enumeration process. This discovery process allows the host to establish other pipes to the device (when necessary) in support of its function.

## ENUMERATION

Most of the magic in USB happens during enumeration. Enumeration is the process by which the host assesses device communication speed, assigns the device an address, and inquires what a device is and what it requires. Because the USB bus is made up of small one-to-one connections, the host can pick and choose with whom it is going to communicate.

When a new device is plugged in, the host (or hub) senses its connection by monitoring the D+ and D- lines. If plugged into the host, it will be detected immediately. If plugged into a hub, the hub will tell the host of a detection the next time the host asks for the hub's status. The host (or hub) can now determine the speed of the new device and make contact using the default address of zero. The first piece of information requested from the device is the device descriptor table. The device responds with a list similar to that of Table 2. (Note that most tables are similar. The bLength field indicates how large the table is. The bDescriptorType indicates the format of the rest of the table.)

The host retrieves information on how large a packet it can handle and assigns the new device its own unique address (leaving the default address zero free for the next newly attached device). The host can now communicate with the device through its new address, using packets of an acceptable size to retrieve the device descriptor and learn about the device's abili-

ties. Based on the device descriptor, the host can request information on each configuration (mode of use) for the device (most devices have a single configuration). By requesting the first 9 bytes of the configuration descriptor, a host can determine how many interfaces (sets of endpoints) the device requires. With this new information, the host rereads the configuration descriptor this time in full, which includes the interface and endpoint descriptors.

Based on the host's acquired knowledge of the device, Windows looks for and installs a driver to match. The driver asks the host to set the device to one of its configurations. The device enters its configured state with its interfaces enabled, signifying that the enumeration has succeeded.

As you can imagine, it only takes a small error in any of the descriptor tables to foil the enumeration process. Windows plays a device connect or device disconnect sound when a USB device is connected or disconnected (failed enumeration plays as a disconnect) so you can tell what is going on. Other than that, don't expect much diagnostic information about the connection or lack of it.

When a device is disconnected from a hub (or host), the hub's status indicates this. The host removes the

device and any downstream devices (if this is a hub with connected devices) from its polling pool.

## COMPLEXITY VS. EASE OF USE

The PC industry has certainly concerned itself with the user. The USB effort will significantly reduce peripheral user support. Rigorous testing before obtaining USB certification ensures the compliance of any device that wants to display the official compatibility logo.

Supporting USB doesn't make the designer's job any easier. In fact, it creates another specialty area, where the average designer must put in long hours to make sense of all the bits necessary to implement a USB interface.

## NOW WHAT?

Got a project that you would like to convert to USB? This seems like a good place to start trying to understand just how USB works. I've got a serial dongle that I designed a few years ago to take serial commands and convert them to X10 signals using the TW-523 as a power line interface (see Figure 4). The TW-523 is an X10 product that gives you isolated access to the utility grid. The TW-523 has two isolated control lines out (zero crossing and 120-kHz gate) and one isolated control line in (120-kHz gate).

To send X10 data, you

must use the zero-crossing input as a reference point to create three potential 1-ms gates (one at each 60° point within 60 Hz of each half cycle starting with the zero crossing). X10 data consists of three gate signals (for logic 1) or the absence of three gate signals (for a logic 0) during a 60-Hz half cycle. Except for data in the start code, the second half cycle must contain inverted data of the first half cycle. The X10 commands are sent as start code, house code, and function code. (Note that the purpose of the three gates is to enable 120-kHz bursts within the TW-523 that are applied to the power line at the zero crossing.

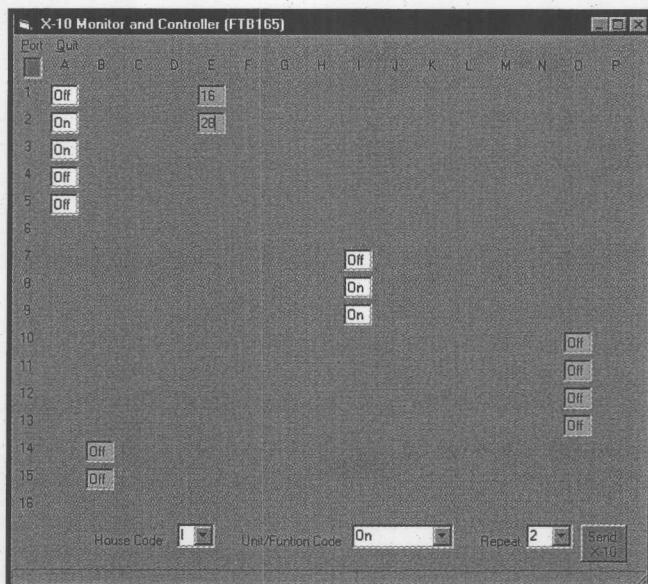


Photo 2—The X10 monitor screen can monitor every X10 address, but it shows only those modules that have received X10 commands. Visible modules can be designated as lamp or appliance modules and therefore properly reflect on, off, and dimness levels. Drop-down boxes at the bottom allow X10 commands to be set and then sent using the "SEND X-10" button.

points for three-phase power, where the other phases are 60° and 120° out of phase.)

To receive X10 data, look for envelopes at zero crossings and decode the data into X10 commands. The serial dongle accepts X10 commands as 3 bytes from a PC application (house code + 128, function code, and repeat code) and translates the commands into X10 power line gating signals. If an X10 command is heard on the power line, 2 bytes are sent to the PC (house code + 128 and function code).

Commands sent to the TW-523 are also sent back to the PC as an operational check. At the time I was investigating the expanded power of microEngineering Labs's PicBasic PRO, which is a basic compiler for Microchip microprocessors.

New X10 commands had been added to the PicBasic's command set, and I wanted to experiment with them. The total basic program text is approximately two pages. Execution uses the XIN command to look at half of a power line cycle for a start code. If one is found, it decodes the data into a house code and

function code. The program sends it out of the serial port. If a start code is not found, the program waits for approximately 5 ms (less than half a line cycle) before looking for a start code again.

The wait loop allows an interrupt to be serviced if necessary (also written in BASIC). A character received via the hardware serial port produces an interrupt. The interrupt routine grabs three characters (house code, function code, and repeat code) and checks to see if they are legal (house code must have most significant bit set), sets a data good flag, and exits the interrupt routine. Back in the main loop, the data good flag enables an XOUT command to be sent. All code for this module is written in BASIC and compiled into assembly code by the compiler.

The application on the PC in Photo 2 is written in Visual Basic; it monitors the serial port for received X10 commands. This application also allows you to send an X10 command to turn on and off an appliance module or control (e.g., on, dim, bright, off) a lamp module. The application monitor screen shows the status of all

255 potential modules (A-Z, 1-16). Individual modules are shown only after an X10 command has been sent or received to or from a module. You can indicate the module's use as a lamp or appliance. This allows the dim and bright levels to be displayed for lamp modules. User-modified descriptions of each module pop up when the mouse hovers over the said indicator.

## HID

In most situations you must write a PC device driver for a particular class USB device. I don't know about you, but that's much more than I wish to bite off at one time. Fortunately, there is the human interface device (HID) class. This was one of the first USB classes to exist, and it has an existing Windows driver. Although the HID class was

designed as a human interface, it can be used for anything as long as the data fits with the class specifications.

Data is exchanged by the host-requesting and device-returning reports via control or interrupt transfers. A report can have multiple transactions of 8, 64, and 1024 bytes for low-, full-, and high-speed devices. Maximum transfers are therefore 800, 64 KB, and 24 MB per second.

Next month we will take a look at using the HID class for this project. Although the present interface is serial, we are not doing any bulk file transferring, so throughput is not an issue. Even when using a low-speed USB device, the specs for this project easily fit within the HID class specifications. ■

*Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for Circuit Cellar since 1988. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.*

## PROJECT FILES

To download the code, go to [ftp://circuitcellar.com/pub/Circuit\\_Cellar/2004/165](http://circuitcellar.com/pub/Circuit_Cellar/2004/165).

## RESOURCES

J. Axelson, *USB Complete: Everything You Need to Develop Custom USB Peripherals*, 2nd ed., Lakeview Research, Madison, WI, 2001.

USB Implementers Forum, [www.usb.org](http://www.usb.org).

## SOURCES

### PIC16F873A Microcontroller

Microchip Technology, Inc.  
(480) 792-7200  
[www.microchip.com](http://www.microchip.com)

### PicBasic Pro compiler

MicroEngineering Labs  
(719) 520-5323  
[www.melabs.com](http://www.melabs.com)

### HIDmaker

Trace Systems, Inc.  
(301) 262-0300  
[www.tracesystemsinc.com](http://www.tracesystemsinc.com)

### TW-523 X10 Two-way interface

[www.X10.com](http://www.X10.com)

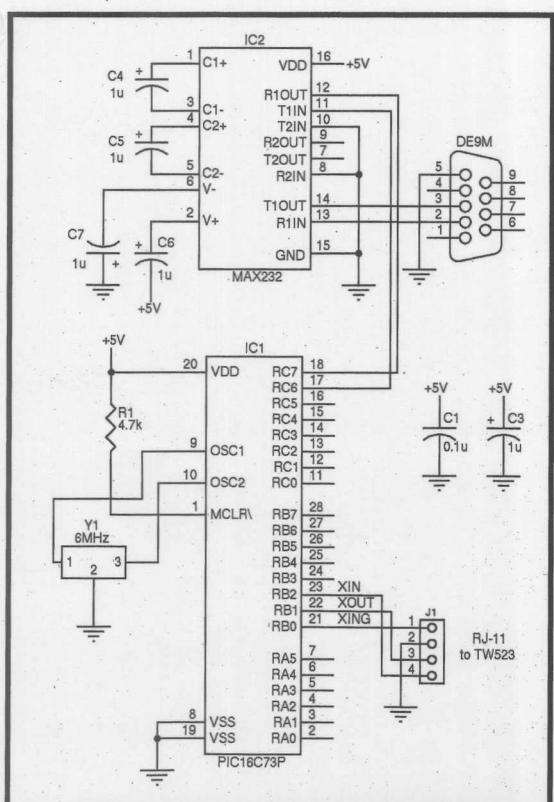
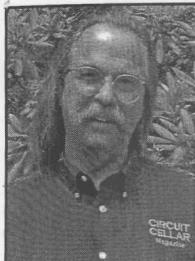


Figure 4—This simple schematic allows a microprocessor to communicate serially with a PC application and to send and receive X10 commands using a TW-523 isolated power line interface.



FROM THE BENCH

by Jeff Bachiochi

# USB in Embedded Design (Part 2)

## HIDmaker Converts an Application

*Jeff makes converting to USB an attractive option with HIDmaker, a highly efficient program that does all the work for you. Creating USB software has never been easier.*

**M**any of you will say they were too young, that they were struck down in the prime of their lives. What a shame, you'll say. They hadn't even reached the age of 50.

Are serial and parallel ports dead? I believe that nothing can really die unless it's forgotten. All of us have family and friends who have passed on, but we keep them alive by sporting their pictures on our walls, visiting gravesites, talking about them, and memorializing them in various public and private ways.

I remember the rumblings that TV would kill radio. (I can't imagine riding in my car without tunes.) But today's technology continues to assure radio's success via satellite. Although my kid's kids probably won't even know what serial and parallel ports are, that's at least a decade away. Although it's getting tougher to foresee what the future holds, I doubt that they will totally disappear anytime soon. It's more likely that they will become obsolete because of non-support in much the same way DOS has been neglected (although still used by some).

"If it ain't broke, don't fix it." I'm a big believer in that statement. But I also understand that much of the world economy's growth is based on the development of new technology and the harnessing of its potential. If we stand still, we run the risk of falling behind. Do you want to run that risk?

### BECOME A USB CONVERT

If you aren't convinced that you need USB after last month's column, you're standing still. Last time, I briefly discussed the reasons for the development of USB and how you can benefit from its

implementation. You observed the physical attributes of the USB connection and how a USB device makes itself known to the host through enumeration. Furthermore, I explained how tiered interconnections allow the host to communicate with each USB device on a one-to-one basis using data packets stuffed into 1-ms frames. I also introduced a previously designed X10 project.

X10 uses the AC power lines in your home to transmit control codes to specially designed outlets that respond to the codes. X10 codes can turn on and off appliance modules (relay-based). They also turn on and off—as well as dim and brighten—lamp modules (triac-based). These modules can be purchased in an AC outlet form factor or as plug-in modules.

The project consists of a serially connected dongle, which translates serial X10 commands into the appropriate control signals to drive a TW523 isolated power line interface. A PC application feeds user-selected X10 commands, as TX serial data, to the dongle while monitoring the RX serial input for X10 commands received from the power line.

Any X10 command can be sent by transferring 3 bytes (HouseCode, UnitFunctionCode, and RepeatCode) to the dongle over the serial interface. An X10 command seen by the TW523 on the power line is returned to the application by the dongle transferring 2 bytes (HouseCode and UnitFunctionCode) back via the serial interface.

To replace the serial interface with a USB equivalent requires hardware and software design changes. These changes, which are not minor, may require choos-

ing a new processor for your design.

### HARDWARE CHANGES

The most visually dramatic changes to a project are physical ones. The area needed for a USB interface is considerably less than that of serial and parallel ports. Besides the connector size shrinking to about 25% of a DB25, only a few discrete parts are needed instead of level shifters or buffers. You can free up a few square inches of real estate on your PCB. Furthermore, you can really luck out if the manufacturer of your microcontroller makes a version with a USB peripheral. More and more manufacturers are jumping on this bandwagon. If not, you may be able to add a USB interface chip (e.g., the Philips ISP110x) to your micro, assuming it has the horsepower and buffers to support all of the necessary USB functions. The great part about using a micro integrated with USB is that all the low-level stuff is done for you.

USB hardware handles detecting and decoding incoming packets. It determines which transactions to ignore (no matching address) and which to react to, flagging the type for endpoint 0. Incoming data is stored in the appropriate buffer or flagged as an error. The number of bytes received and the data-toggle state is stored. The hardware calculates and compares CRC values, and takes the appropriate action on errors. It automatically sends any necessary handshaking to the host and triggers an interrupt for the firmware to take action. Outgoing data is encoded for transmission along with the byte count and data-toggle code.

The hardware also calculates a CRC and appends it to the data packet. After

receiving a handshake from the host, it triggers an interrupt for the firmware to take action. I'm sure you'll concede that this functioning is something to be avoided if possible. You can sidestep this entirely by designing with a microcontroller that has USB support.

## SOFTWARE CHANGES

Data transmission via a serial port requires the conversion of data through hardware or software into asynchronous or synchronous bitstreams. The two devices are configured for the same data format so that what goes in one end comes out the other end. They may use hardware, software, or no flow control. These factors must be agreed to prior to any communication; for the most part, they cannot be identified through the connection itself.

Data transmission via parallel ports moves data in byte-sized transfers using eight parallel data paths. Although potentially 10 times faster than serial, hardware handshaking is done on a byte-by-byte basis, which slows everything down.

The original parallel port is unidirectional and—discounting the 5 bits of status inputs, which are often used for inputting nibble data—is not meant to be a bidirectional device. This led to the bidirectional port (SPP), the enhanced parallel port (EPP) for speedier nonprinters, and the extended capabilities port (ECP) for speedier printing devices. Which do you have? Truth be told, unless you are using an older system, you probably have support for all of them, but it is still half-duplex.

Although similar to synchronous serial, a USB host can obtain everything it needs to know about a device through the connection. Therefore, you don't have to know anything about a device just to make a connection to it. It's easy for you (as a user), but tough on the designer.

A special device descriptor is the key to each device. This is the biggest stumbling block for a designer. The hardware handles much of the work. For the most part, after putting together the device descriptors, it is simply a matter of managing your data in and out of the endpoint buffers.

Handling this data takes some thought, however, particularly because of the way USB operates. Your project

may be used to sending data whenever necessary to the application. Under USB rules, the host must ask for data before a device can send it. You're in trouble unless you have the resources to hold the data until it's asked for.

Last month I covered the designed data throughput of USB. Now is a good time to cover the four types of transfers (control, interrupt, bulk, and isochronous) because each has different throughputs (see Table 1).

The all-important enumeration process is handled with control transfers. The host allocates between 10% and 20% of a frame to control transfers. Although all enumeration begins with endpoint 0, control transfers are not limited to endpoint 0 or enumeration. A control transfer can be used at any time as defined by any class (or vendor).

Don't be fooled by its name, the interrupt-type transfer does not cause an interrupt; instead, it is guaranteed to occur within a certain amount of time (more like interrupt latency). The interrupt transfer is used where you don't want to miss a key press, mouse move, or any other real-time interaction event. (The Windows predefined HID driver supports this mode.) This type of transfer also requires separate in and out pipes for data. Although attempts by the host are guaranteed, it's up to the device to be ready and to ensure any throughput. The latency time is defined in the device's descriptor table.

Bulk transfers are similar to interrupt transfers, except there is no guarantee on when the data will start or finish. They are plugged into frames whenever there is room. This type of transfer also requires separate in and out pipes for data. With no USB traffic, a bulk transfer can fill the full frame. However, if other USB transfers fill the frames, a bulk transmission is held off. Bulk transfers are generally limited to functions that aren't time critical (e.g., printing and file transfers).

The isochronous transfer is similar to an interrupt transfer in that there is a guarantee. Here the isochronous transfer requests a specific bandwidth. The host must calculate what bandwidth it

Transfer type	Maximum data transfer rate per endpoint (KB/s) (data payload/transfer = maximum packet size for the speed)		
	Low speed	Full speed	High speed
Control	24	832	15,872
Interrupt	0.8	64	24,576
Bulk	N/A	1216	53,248
Isochronous	N/A	1023	24,576

Table 1—A wide range of throughput is available depending on the bus speed and transfer type that you select for your design. Note that bulk and isochronous types can't be used with the low-speed bus interface.

has to offer before it can accept and configure an isochronous pipe. Success depends on which other devices are already on the bus. This type of transfer also requires separate in and out pipes for data. You can see that allocating bandwidth ensures the data gets there; however, there is no plan for resending corrupted data. Therefore, this type of transfer is used where the application can tolerate small errors (streaming audio). The bandwidth requirements are defined in the device's descriptor table.

## DEVICE CLASSES

If you could gather all the USB devices out there (as well as those that haven't been designed yet) and separate them into groups with similar characteristics, you could define classes of devices, which would include the following: the HUB device, audio device, chip/smartcard interface device, communication device, content security device, device firmware upgrade, human interface device (HID), IrDA bridge, mass storage, printer, and imaging. The list is continually growing.

As you can imagine, generic drivers don't hack it for most peripherals. There is the necessity for special drivers to be written to handle each of these devices, even those within the same class. After the host has received a device descriptor table through enumeration, it can search its .INF (or other files) for a matching class and device driver. If necessary, the host asks you for the appropriate drivers. (That's if the device is new to the system.)

Windows has a generic driver for the HID class. (Try searching for hid-dev.inf; it can be viewed with Notepad.) Because HIDs have a generic class driver, designers can make

their lives less complicated by trying to design for this class.

## HUMAN INTERFACE DEVICE

As its name suggests, a HID has to do with devices that interact with you. This covers numerous input and output devices (e.g., keyboard, mouse, steering wheel, rumble pack, and joystick). Many monitors have an embedded HID device, which is used for screen configuration rather than video. In actuality, a device does not need to interface with you to be a HID. If your device's data requirement fits within the HID class's specifications, you're golden.

My conversion project fits within this specification. I needed to be able to transfer 3 bytes out of the PC to the interface or 2 bytes into the PC from the interface.

X10 transmissions are slow because they are based on 60-Hz zero-crossings. X10 commands require about 0.5 s (22 60-Hz cycles) to be completed on the power line, so you don't need to be worried about exceeding available frame times. As far as sending data from the PC, the interface is designed to transmit an acknowledgement of received data (loop back); therefore, loss of data because the interface is busy is handled by the host application expecting looped-back data.

On the interface side, I had to create the appropriate descriptor tables and change the interface application code to make use of the USB hardware in the micro. Assuming the generic HID driver is usable on the host (PC), I had to change the data exchange routines in the application from serial to USB.

### HIDmaker

Trace Systems looked into the future and built a product to get you up and running quickly with USB.

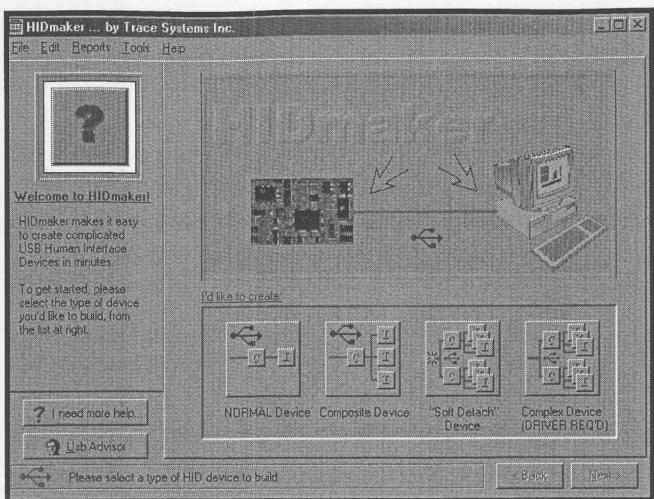


Photo 1—HIDmaker simultaneously generates programs for the peripheral device and the PC. The programs can communicate with each other over the USB via the HID class. Even complex interfaces can be modeled with HIDmaker.

HIDmaker was designed for use with the HID class, so no device driver programming is required. You get all the benefits of USB with less work than RS-232 serial because it does the hard work for you!

HIDmaker creates two documented programs—one for the PC and one for your microprocessor—that already communicate with each other using USB. The programs, which are built around the data that you describe, are written for Microchip USB PICs in your choice of environments: PicBasic Pro, Microchip MPASM, CCS C, or Hi-Tech PICC. On the PC side, HIDmaker supports Visual Basic 6, Delphi, and C++ Builder. (These environments are the first to be supported by HIDmaker with other microcontrollers and man-

ufacturers to follow.) It's a perfect fit for this project because I originally used a PIC (and PicBasic) for the device and Visual Basic for the application.

Photo 1 shows the opening HIDmaker screen. The program is set up in the familiar Wizard style. After you add the required information to each page, click the Next button to move on. You can save the project file and exit any page. Loading the project file brings you back to where you left off.

To begin the HIDmaker process, select a device type.

This tells HIDmaker how complicated your new device is. The device in this project is simple. More complex devices have multiple, independent types of functionality (a combination keyboard/mouse).

On the second page, you name your project and select a location for the files that HIDmaker will produce. You also add information like the vendor ID and project ID, which are required by Windows to identify your device. Any device sold commercially must have a vendor ID from USB Implementers Forum.

Notice the various boxes labeled "Use in F/W." If checked, the optional information can be placed in device firmware by HIDmaker to be discovered through enumeration by the host.

HIDmaker offers two types of help. The "I need more help..." button provides context-sensitive help for using the controls that are currently visible on the screen. The USB Advisor button gives advice about appropriate inputs, what the inputs mean, and how they relate to the process.

The third page gets down to the nitty-gritty. I've already indicated that this is a normal device. As such, it has a single configuration. As you can see in Photo 2, most entries already have selections;

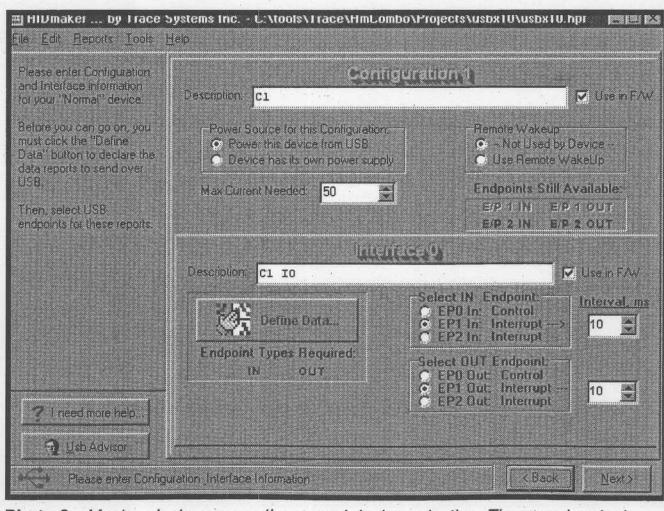


Photo 2—Most projects can use the normal device selection. They need a single interface defined. Complex devices require you to define an interface scheme for each configuration.

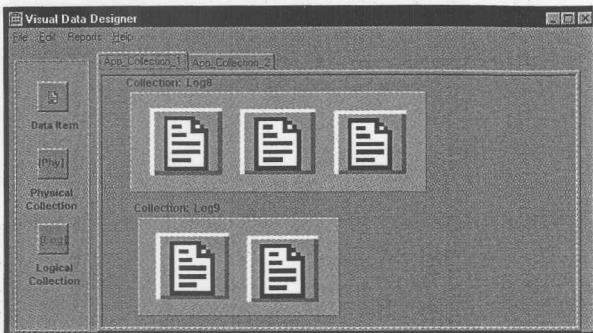


Photo 3—Data items can be placed in logical or physical collections. Collections aid in understanding how the data items are used but do not affect how the data is actually handled.

these are the most common usages, but alternatives can be selected.

If you want the USB interface to supply power to the device, you must also indicate the device's current consumption. Because the total current available through the USB interface is limited, the host uses this value during enumeration to determine if there is available current to allocate to the device.

As you know, low-speed devices are limited to two kinds of transfer types, control and interrupt. I chose to use interrupt transfers to ensure periodic transfers. Because I need data to go in both directions, I choose EndPoint1 IN and EndPoint1 OUT. The host creates separate pipes (virtual connections) between it and the device endpoints based on this information. Note that you also get to indicate how often you want each transfer to take place.

Only one more bit of information must be defined: the data that will be transferred. Clicking on the Define Data button brings up the fourth and final page (see Photo 3). The data for this project consists of three values going to the device (using out endpoint 1) and two values going to the PC (using in endpoint 1). This page has three buttons on the left and a large drawing area. Use this page to create data items—one for each variable used in each pipe.

The two lower icons on the left—Physical Collection and Logical Collection—help you indicate which data items are grouped together. They can be placed in the drawing area as placemats for the data items. For instance, you may want to use two Logical Collection placemats, one for the three out data items and one for the two in data items. These don't

actually affect how the data will be defined; they merely aid in documentation. For each item of data, click the Data Item button and place an icon in the drawing area. Now all that's left is to define each item.

When the mouse hovers over a data item, a pop-up shows a description of the item's configuration. By double-clicking on the item, you'll see that data item's

editable properties (see Photo 4). I changed each data item's name to the variable name I use in the applications (again, to help with documentation).

The data type must be selected based on the endpoint, either input or output, for this project. The usage info consists of a combination of two numbers: a Usage Page number and a Usage ID number, which is the number of an item in the Usage Page. A Usage ID represents some particular function or feature of a device, and the Usage Page is a collection of Usage IDs that belong in the same category. The HID usage table's specification defines and documents nearly 1600 combinations of Usage Page and Usage ID. Click the Browse Usages button to pick from a list of all the standard usages that were recently defined as well as a modest number of Vendor Defined usages.

It is important that every data item you define has a distinct Usage ID; this is the only way that the HID class, as implemented by the Microsoft Windows HID API, can identify and locate

your data items. I chose to use the vendor-specific User Page (0xFF, the default) and User ID's 0x00-0x04 for the five data items. Notice that you can define the maximum and minimum values for each data item. This, in turn, determines the number of bits necessary to define your item.

My HouseCodeIn/Out items require 8 bits. The UnitFunctionCodeIn/Out and RepeatOut items only require 5 bits each. (The transferred USB data packet contains data items packed together to save every bit possible.)

Although this project is fairly simple and the default values are used in most cases, you can see that the data properties offered are both flexible and complex. HIDmaker's guidance isn't fully appreciated until you look at the output generated by compiling and building sample applications based on all the input that you complete.

## HIDmaker DEVICE OUTPUT

After you have fully defined your device, its data, and its interface, HIDmaker can work its magic. Select the environments that you want HIDmaker to generate output for (I used PicBasic Pro and Visual Basic), and in seconds you'll receive a list of all the output files for both the peripheral and the host. You can peruse the files before exiting HIDmaker. You may download the files from the *Circuit Cellar* ftp site.

On the peripheral side, HIDmaker generates two main files (nine files in all for this project). First, *descript.asm* holds all of the descriptor tables defined from the information entered in HIDmaker. Then there is *USBX10\_M.bas*, which is this project's PicBasic Pro application file. The following four files are project-independent. Three Microchip files—*usb\_ch9.asm*, *usb\_defs.inc*, and *hid-class.asm*—support the PIC USB parts. One file, *usbdesc.asm*, lists a number of include files specific to PicBasic Pro.

The *descript.asm* file, which is cer-

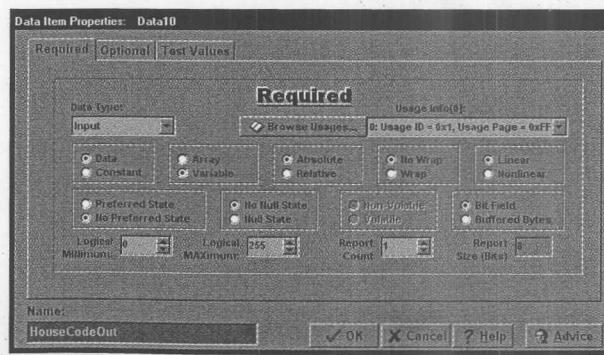


Photo 4—Although each data item must be fully defined, many default selections can be used. Take some time to make sure your data will be handled with the respect it deserves.

tainly the most important, contains all of the tables the host reads during enumeration to gain the ultimate knowledge of your device. For this particular project, it contains a device descriptor table, configuration descriptor, interface descriptor, HID descriptor, endpoint descriptor (for EndPoint1In), endpoint descriptor (for EndPoint1Out), and report descriptor. That's not to mention the string descriptors as defined in HIDmaker (e.g., language ID, manufacturer, product name, serial number, and other option strings).

Using Microchip's IDE and PicBasic Pro, I compiled the project files into a hex file, which is used to program a PIC16C745. Let's take a quick look at what HIDmaker built for this simple application.

HIDmaker creates a USB interface capable of moving defined data. In this case it does this by setting up an application to pass three data items from the host and collect two data items from the device using a USB connection. After some initialization, the application proceeds to a main loop where the USBIN command checks to see if there is any out endpoint 1 data available. If so, it retrieves and unpacks the data. Otherwise, it goes on.

In the second part of the loop, the out endpoint 1 data items are set to test values (initially defined in the define data section of HIDmaker), and

a flag is set to indicate that data is ready to send. The data is packetized and the USBOUT command is executed, which sends the data to the USB in endpoint 1 buffer if it is free. (Otherwise, it tries again.) Then it's back to the main loop.

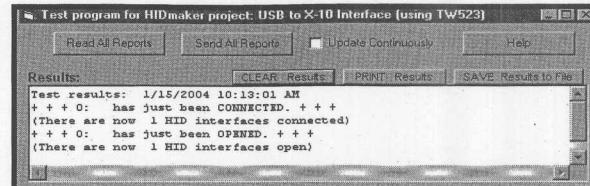
Notice that this application passes data but there is no apparent check on the data. This could be easily modified in this case so that the data that comes in is used for the data that goes out. But this is not necessary because there are ways to see the data inside the PIC. At this point, if everything was done correctly, the PIC with the necessary USB circuitry will enumerate when it's plugged into a USB port on the PC (see Figure 1).

## HIDmaker HOST OUTPUT

On the host side, HIDmaker creates three main files and a bevy of support files (11 files in all for this project). USBX10\_M.frm is the main program module. Adding this form to a project enables the HIDagentX object, which is used for a USB interface. It is similar to using an MSComm object for a serial interface.

The USBX10\_L.cls class module library file handles the variables and functions that make direct interfacing with the HID system calls unnecessary. The OneHidVar.cls class module file represents one USB HID data item or variable with all its attributes. Additional modules are created by HIDmaker, PC\_Consts.bas (a file of expected device strings) plus error checking, and context help support files.

You can create a project .EXE file for the HIDmaker-created application using Visual Basic IDE, or you can run it in Debug mode from the IDE. Photo 5 gives you a quick picture of what is going on. The application will not run if it doesn't find a matching device enumerated on the USB bus. Assuming it does, the application pops up a window and shows that it has found a device. You can use the Send All Reports and Read All Reports buttons to



**Photo 5—**HIDmaker creates device and PC applications that complement one another. Here the PC application demonstrates that it has succeeded in providing a working USB connection by allowing you to pass your defined data to and from the PC.

test the communication to and from the attached device. The transferred data, which is hard coded into both applications (the host PC and the device microcontroller), originates from the HIDmaker define data steps.

Unless you can see into the microcontroller, you won't know if the correct data has arrived. However, you will see the data arriving from the device displayed in the host application program. At that point HIDmaker has proven its success. You can take these applications and integrate them with your own code. But wait, HIDmaker is still of use.

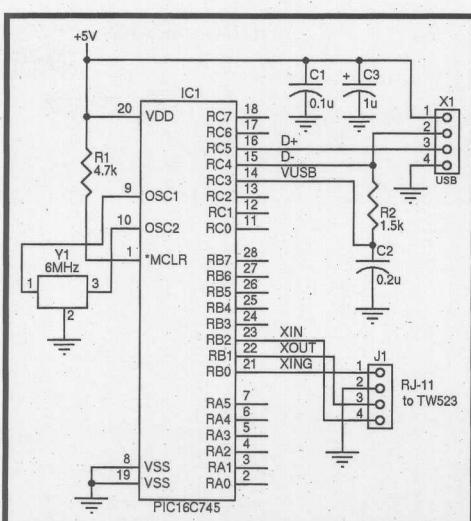
## OPTIONAL TOOLS

Remember how I said that you couldn't see inside the micro? Well, if Microchip had its act together, they would have the USB devices available in flash memory and the ICD in-circuit debugger could be used to look at the microcontroller's registers.

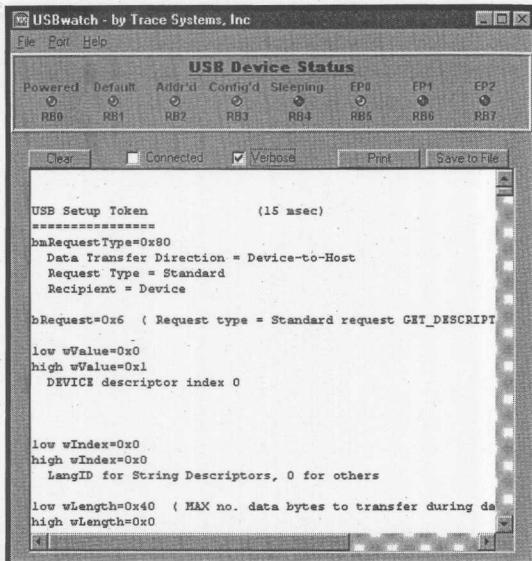
Trace Systems has added hooks within its code to allow messages to be output through a serial port on the microcontroller. (Isn't it ironic to use a serial port to debug its replacement?) You can use this technique to log messages (including any values) at any point in the application, so you can see the values received from the application. In fact, if for some reason your device doesn't enumerate correctly, this can provide you with feedback about the failure's location.

Enumeration failure can be the biggest source of headaches for a designer. Although HIDmaker generates working code for you, if enumeration fails, HIDmaker offers suggestions about potential problems.

USBwatch is a terminal-style application that interprets tokens sent by the microcontroller out of the debugging serial port. Tokens are used to reduce serial traffic to a minimum. The



**Figure 1—**The PIC16C745 microcontroller has USB support for version 1.1. The interface is simple and can be powered from the USB bus. The microcontroller can apply an internal phase-locked loop to operate at 24 MHz using an external 6-MHz crystal.



**Photo 6**—The USBwatch interpreter works in conjunction with a serial port on the microcontroller. Embedded commands allow you to see what's happening inside the microcontroller. You determine the information reported by enabling the hooks within the HIDmaker-generated code.

USBwatch application expands the tokens back into real English text. Note that there are a number of indicators above the text window in Photo 6. These help determine the status of enumeration and the present endpoint in use.

From the PC side, you can investigate all USB devices using the AnyHID application, which shows all of the USB devices (except hubs and the system's keyboard and mouse). When you choose a device, AnyHID will tell you all about it by reading the descriptor tables from the selected device (see Photo 7). Because the descriptors contain information about each data item, AnyHID is presents you with the opportunity to configure a data packet to send to the device and to receive and interpret requested data.

## TIME TO CONVERT

HIDmaker's microprocessor test application code is well documented and uses embedded comments to suggest where you might add your application

code. PicBasic's USBIn1 command retrieves any received USB output data packet while HIDmaker's code unpacks the data and sets the HandleEp1Rcv flag when data is available to you from the host. In a similar fashion, you set the EP1XmtDataReady flag when you have data to send to the host. HIDmaker's code packs your data and uses the PicBasic USBOut1 command to send USB packets to the host when it requests data.

On the PC side, I opened my serial-based application in Visual Basic. All of the modules and class modules that HIDmaker created for the test application are added to the project. The MSComm object on my main form

(Form1) was replaced with the HIDAgentX object. The FRMPROPS.FRM form was deleted because this is the MSComm popup form that allows you to set the serial port properties. The project was saved as USBX10.

Next, the code in the USBX10\_M.frm form that produced the test application form had to be eliminated or the necessary code had to be copied from

the USBX10\_M.frm form to Form1. I chose the latter.

Gone are four routines that deal with the serial port, opening and closing the serial port, and sending characters to and getting characters from MSComm. In their place HIDAgent's routines become the low-level engine that does all the hard work of opening HID devices, accessing HID data items, and packing and unpacking reports. After HIDAgent has detected, identified, and bound itself to an enumerated device, its runtime events become accessible.

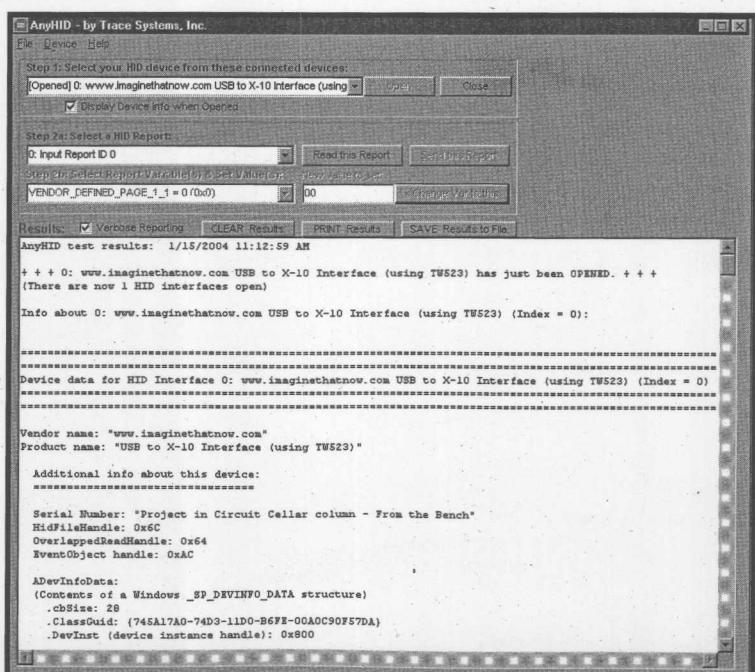
The Sub ReadRpts() routine periodically transfers data items from an input report. I placed code within this routine to test for new data, manipulate it, and perform an update on the form using the received X10 command (new data). To send an X10 command, I used the CmdSendX10\_Click() event to assign values to the data items before requesting a WriteAllReports.

## SALVATION AT LAST

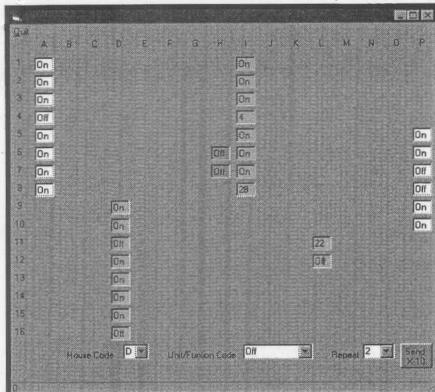
The finished application shows little sign of change (see Photo 8). Two areas of concern popped up while using HIDmaker to help convert the application. First, thanks to Microsoft's wisdom, all data is treated as signed unless specifically told not to. This results in a popup asking if this should be ignored

each time the program is executed. The Windows routine is avoided by setting the Scale mode to a value of one (unscaled conversion only) in the OneHidVar.cls class module file.

The second area has to do with the way I designed the USB device. USB likes to have data available from a device at all times. This can cause a problem for your application if you have new data that is the same as the old. If it is the same, how will the application determine if the data from the device is old or new? Instead of adding complexity using some kind of flag



**Photo 7**—AnyHID is a peephole into USB devices enumerated by your PC. You can interrogate each device and actually send and receive data to and from the device.



**Photo 8**—With the conversion now complete, the Visual Basic application originally written using a serial port interface operates using the new USB interface. Your customer may never notice the difference.

exchange, I don't provide data to the USB port unless it is new. So, most of the time when the host asks for data, the device will not answer and the host will timeout after a default (default = 5000, 5 s). This causes some rather nasty application lags during periods of no new data (most of the time). Shortening the USB timeout property to 100 ms in the CreateHidObjects routine eliminated it.

Using USB for I/O is no longer as easy as writing to a port. To become a user-friendly interface, USB has made the designer's job extremely complicated. Luckily, manufacturers are realizing they need to help reduce the burden. Trace Systems's package shortens the learning curve. HIDmaker, along with its optional tools AnyHID and USBwatch, provides the support needed to make USB come alive for the designer. Trace Systems knows what you need, and it supplies plenty of support documentation like the USB, HID class, and usage table specifications. ■

*Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for Circuit Cellar since 1988. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.*

### PROJECT FILES

To download the code, go to [ftp://ftp.circuitcellar.com/pub/Circuit\\_Cellar/2004/166](ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2004/166).

### RESOURCES

J. Axelson, *USB Complete: Everything You Need to Develop Custom USB Peripherals*, 2nd ed., Lakeview Research, Madison, WI, 2001.

USB Implementers Forum, [www.usb.org](http://www.usb.org).

### SOURCES

#### PIC16C745 Microcontroller with built-in USB support

Microchip Technology, Inc.  
(480) 792-7200  
[www.microchip.com](http://www.microchip.com)

#### PicBasic Pro

microEngineering Labs  
(719) 520-5323  
[www.melabs.com](http://www.melabs.com)

#### HIDmaker

Trace Systems Inc.  
[www.tracesystemsinc.com](http://www.tracesystemsinc.com)

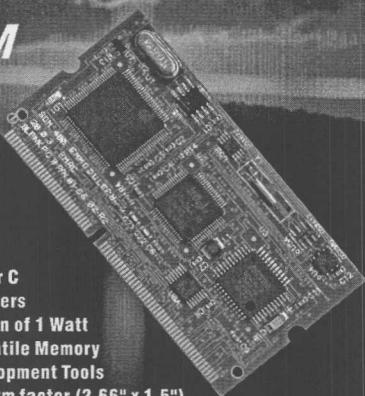
#### TW523 X10 two-way interface

[www.X10.com](http://www.X10.com)

## TINI® Webserver Internet Appliance Engine

### SOM-400EM

- DS80C400 Processor
- IPv6 Fast Ethernet
- 3 Serial Ports & SPI
- SODIMM Bus Expansion
- Uses a bash shell subset
- 1 MB Flash & 2 MB RAM
- CAN 2.0B & 1-Wire® Ports
- Programmable in Java™ or C
- Telnet, FTP, and HTTP Servers
- Typical Power Consumption of 1 Watt
- Real Time Clock & Nonvolatile Memory
- Robust, Free Java™ Development Tools
- Small, 144 pin SoDIMM form factor (2.66" x 1.5")
- Carrier/Socket Board & Optional Power Supply Available



The SOM-400EM is a System on a Module, based on the Dallas' Tiny InterNet Interface (TINI®) Processor. This 8051 code-compatible processor makes it extremely easy to create a smart Network / Internet capable device. The SOM comes complete with Embedded Java OS and File System. Write sophisticated network applications in days instead of months. Single unit pricing starts at \$105. For additional information visit, <http://www.emacinc.com/som/>.

Since 1985  
OVER  
19  
YEARS OF  
SINGLE BOARD  
SOLUTIONS

**EMAC, inc.**  
EQUIPMENT MONITOR AND CONTROL

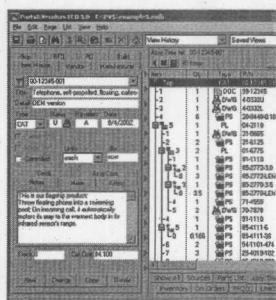
Phone: (618) 529-4525 • Fax: (618) 457-0110 • Web: [www.emacinc.com](http://www.emacinc.com)

## Parts List Software for Engineers and Designers

- Easily create and manage multi-level parts lists for products in development...and after.
- Track sources for items with multiple price breaks.
- Calculate product costs at any quantity.
- Launch CAD, viewer or browser from any item.
- Automatically generate RFQs or POs.

### New Version 5.0

- New Report Layout Editor customizes reports/labels.
- New Connection to QuickBooks 2002/2003 Pro simplifies accounting (us version only).
- New Multi-currency for foreign suppliers eases exchange rate calculations.



## Parts & Vendors™

Visit [www.trilogydesign.com](http://www.trilogydesign.com)  
and download our FREE DEMO.

For Windows  
98/NT/Me/2K/XP  
3 Editions,  
starting at  
\$99 + s/h

Or, Call 800-280-5176

530-273-1985 Fax 530-477-9106  
P.O. Box 2270, Grass Valley, CA 95945